THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: John Chun-Sing Lam

TITLE OF THESIS: On Optimizing Processor Bounds of Certain Parallel Graph Theoretical Algorithms

DEGREE FOR WHICH THESIS WAS PRESENTED: Master of Science

YEAR THIS DEGREE GRANTED: 1979

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

THE UNIVERSITY OF ALBERTA

ON OPTIMIZING PROCESSOR BOUNDS OF CERTAIN PARALLEL GRAPH THEORETICAL ALGORITHMS

by

© JOHN CHUN-SING LAM

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

FALL, 1979

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled "On Optimizing Processor Bounds of Certain Parallel Graph Theoretical Algorithms" submitted by John Chun-Sing Lam in partial fulfilment of the requirements for the degree of Master of Science.

ABSTRACT

Under an idealized model of parallel computation, parallel graph theoretical algorithms are studied using adjacency matrices as the representation of graphs. Two optimal algorithms for computing $A(n) = a(1) \circledcirc a(2) \circledcirc \ldots \circledcirc a(n)$ are presented whose time bounds are proven to be equal to the theoretical lower time bounds in both bounded and unbounded parallelism, where $\circledcirc$ is binary associative. The technique for processor optimization is demonstrated through an example on graph problem: finding the connected components of an n-node undirected graph. Efficient parallel graph algorithms for detecting the existence of negative cycles, finding the strongly connected components, verifying unilateral connectivity and acyclicness of an n-node directed graph are developed. Each algorithm runs in $O(\log n(n^{2.81})/P)$ time with $P \leq n^{2.81}/\log n$ processors.

ACKNOWLEDGEMENTS

I would like to thank Dr. I-Ngo Chen, my supervisor, and Dr. Francis Chin for their generous support and guidance throughout the development of this thesis. I am also grateful for the valuable comments and careful reading of the thesis by Dr. H.L. Abbott. I would also like to express special thanks to Jim Achugbue, Elton Ho and Jim Lamont for their valuable discussions and laborious proof-reading of the manuscript.

A special note of thanks must go to my parents and my fiancee, Eva, to whom this thesis is dedicated, for their patience, encouragement and support.

I gratefully acknowledge the financial support from the Department of Computing Science in the form of teaching assistantship, and the National Research Council of Canada in the form of research assistantship, through Grant A7133 to Dr. I-Ngo Chen.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Because of the generality in their mathematical structure, graphs have proven to be a useful abstraction wherever discrete objects and binary relations are dealt with. They have aided analysis in biology, chemistry, computing science, electrical and civil engineering, operations research, linguistics, sociology and many more. These uses of graphs require many graph theoretical algorithms. Consequently, since the early sixties, a great deal of effort has been directed to producing and analysing graph theoretical algorithms. However, the graphs resulting from many real-life problems are often so large that the worst case complexity analysis is extremely pessimistic. This necessitates the development of more efficient and economical algorithms.

To tackle this problem, three approaches have commonly been used in the literature. The first technique is to search for better algorithms with respect to time. The second method is to improve the circuit switching speeds technologically. The third way is via parallel processing. In particular, parallel computers which are capable of performing several independent operations concurrently are designed which tradeoff the amount of hardware for computation time.

Tremendous progress has been made along the first two lines for the past two decades. But optimal algorithms can be improved no more in the first approach, yet may still be unwieldly for large graphs. And as improvements in switching devices and miniaturization rapidly reach their physical limits, it is evident that any significant improvement in processing speed must be obtained by the concurrent processing of a number of operations. For the third approach, various multiple-processor systems have been proposed and constructed. As a result of the recent revolution of microprocessors and the steadily dropping hardware prices, large-scale parallel computers with as many as 2^{14} to 2^{16} processors have become feasible [33,40].

Since a parallel algorithm may be obtained by recognizing the inherent parallelism of a sequential algorithm, many people think that parallel computations are mere extensions of sequential computations. This intuition is not always correct. Adopting an efficient sequential algorithm, such as depth-first search, on a parallel computer is clearly far from optimal. Conversely an inefficient sequential algorithm, such as finding transitive closure by matrix multiplication, can lead to an efficient parallel algorithm. Moreover, to recognize the inherent parallelism of many sequential algorithms is not straightforward. Therefore it is a significant and important problem to design efficient parallel algorithms for the parallel computers. However, it is only recently that more

attention has been paid to the development of parallel graph theoretical algorithms [2, 12, 13, 14, 19, 21, 28, 37].

In this thesis, we concentrate our study on graph theoretical algorithms with polynomial complexities (i.e., time, processor and storage complexities are bounded by a polynomial in the size of the problem). The basic idea is to identify the most important parts of an algorithm (i.e. most time or processor consuming), and put most effort into optimizing those parts. Based on this idea, the processor bounds on most of the existing parallel graph algorithms (see Table I) are improved and in some cases, we show that the processing power is optimally utilized. Also new parallel algorithms for several graph problems are formulated.

To prepare for the discussions in the following chapters, the background material is described in this chapter. In Section 1.1, the classification of parallelism, the theoretical model of parallel computation and measurements of parallel complexity are presented. Definitions and representation of graphs are given in Section 1.2. Previous results are briefly summarized in Section 1.3.

1.1 Model and Measurements of Parallel Computation

Existing parallel computers, such as matrix computer ILLIAC IV, pipeline computer CDC STAR 100, associative computer Goodyear STARAN, and multiple-processor system UNIVAC 1110, are widely different in their architectures and characteristics. A special issue of ACM Computing Surveys [15] provided an excellent survey of these parallel computers. Parallel computers have been categorized in many approaches [5,16,42]. Broadly, they can be classified by the following criteria:

1. General or special-purpose system.
2. Synchronous or asynchronous executions of the operations.
3. Bounded or unbounded parallelism. The former, also called K-parallelism or K-computation, refers to having a fixed number of K processors available while the latter has an infinite number of processors available.
4. Single or multiple instruction streams. In the first case, all processors either execute or ignore the current instruction broadcasted by the control unit, though using different data and depending on a local on/off switch. This is called Single Instruction Stream-Multiple Data Stream (SIMD). In the second case, processors may perform different instructions yeilding the Multiple Instruction Stream-Multiple Data Stream (MIMD). The terms SIMD and MIMD are due to Flynn [16].

From the classification, one can easily observe that

parallelism has been defined on various levels ranging from the bit to the system level. At a lower level, it may refer to arithmetic operations on the whole word rather than on one bit at a time. At a higher level, called the algorithm level, it may refer to parallel executions of independent statements composing the program. Finally, at the highest level, it may refer to concurrent processes of two or more conceptually distinct and independent programs. In what follows, our attention is limited to algorithm level.

Moreover, the structure and performance of an algorithm will not solely depend on the problem at hand, but also depend on the advantages and limitations of the computer. In order to remove these restrictions and exploit the inherent parallelism, a more flexible and more powerful theoretical model is essential which is discussed in the following section.

1.1.1 Model of Parallel Computation

A number of models of parallel computation have been proposed by various authors [2, 11, 13, 19, 20, 27]. The idealized model used in this thesis is similar to Csanky's model [13], which has been widely used, and which satisfies the following assumptions:

(1) An arbitrary number (generally bounded by a polynomial in the size of the problem) of identical processors and

a sufficiently large memory accessible by each processor are available.

(2) Instructions are always available for execution as required and are never held up by the central control unit. Processors are synchronized and all instructions executed in parallel are identical (SIMD).

(3) Initially, the input data is stored in the memory. At any time, two processors may read from but must not write into the same memory location simultaneously.

(4) Each processor is capable of performing any one of the arithmetic, Boolean and comparison operations. At any time, each processor may fetch its operands from the memory, perform an operation and store the result in the memory in one step called a time unit.

(5) No memory or data alignment time penalties are incurred.

In reality, all parallel algorithms must deal with the complex problems of data manipulation, storage allocation, memory interference and interprocessor communication. An ideal interconnection network for communications among the n processors and memory should directly link each processor to every other processor and memory, but this is far too costly for large n. To remedy this problem, memory is partitioned into units and different restricted networks have been proposed [38] which cause possibly significant communication delays. For instance, Gentleman [18] pointed out that in the

two-dimensional rectangular grid network, communication delay is the limiting factor for matrix manipulations.

Nevertheless, in some problems, these delays can be minimized by careful redistribution of data and reindexing processors, without significantly affecting the computation time [1, 6, 25, 41].

The unbounded parallel algorithms presented in the following chapters can be transformed to produce the time bounds on the corresponding bounded parallel algorithms. This is demonstrated through an example in Section 3.1.3, which is to find the connected components of an undirected graph, where both bounded and unbounded complexities are given. Furthermore, it is quite easy to extend the algorithms presented here to MIMD systems and the processor requirement may consequently be reduced. Thus, we feel that algorithms developed under such an idealized model will be influential in creating algorithms for realizable parallel computers.

1.1.2 Measurements of Parallel Complexity

The parallel time complexity of the computation is the least number of time units necessary to produce the result rather than the total number of operations performed by all processors. Therefore the processor requirement is regarded as an important parameter. It is of considerable practical interest to evaluate the effectiveness of parallelism. In

order to measure the performance of a parallel algorithm, the size of the problem (e.g., the number of nodes and edges of the graph) has been chosen as a parameter to determine the time, processor and storage requirements.

Let $T(P)$ be the number of time units required by a parallel algorithm using $P \geq 1$ processors. The speedup of the P-processors computation over the corresponding uniprocessor computation is defined as $S(P) = T(1)/T(P) \geq 1$, and the efficiency which indicates the utilization of the processing power is defined as $E(P) = S(P)/P \leq 1$.

Our first objective is the construction of efficient parallel graph algorithms ideally exhibiting linear (in P) speedup. This situation is realized only when all P processors are loaded in each time units. However, linear speedup is seldom achievable. In practice, the "fast" speedup is $S(P) = O(P/\log P)$, which is acceptable although less than linear. For those existing parallel algorithms with nonlinear speedup, our second objective and the main achievement of this thesis is the minimization of the processor requirement without increasing the time complexity by more than a constant factor.

1.2 Definitions and Representation of Graphs

1.2.1 Definitions

An undirected graph $G = (V,E)$ consists of a finite, non-empty set V of n elements called nodes and a set E of m unordered pairs of nodes called edges (v,w). $G' = (V',E')$ is a subgraph of G if $V' \subseteq V$ and $E' \subseteq E$. A directed graph $D = (V',E')$ is defined similarly, except that edges are ordered pairs of nodes; v is called the tail and w the head of the edge (v,w). If $(v,w) \in E$, nodes v and w are adjacent and edge (v,w) is incident on nodes v and w; whereas if $(v,w) \in E'$, v is said to be adjacent to w while w is adjacent from v, and the directed edge (v,w) is incident from v and to w. The adjacency matrix of a graph is an nxn matrix A of 0's and 1's, where the (i,j)-th element, $A(i,j)$, is 1 if and only if there is an edge from node i to node j. The weight matrix of a weighted graph is an nxn matrix W such that $W(i,j) = w(i,j)$ if there is an edge from node i to node j, and $W(i,j) = c$ otherwise, where $w(i,j)$ is the weight of edge from node i to node j, and c is usually 0 or ∞ which depends on the interpretation of the weight and the problem to be solved. The transitive closure A" of an nxn adjacency matrix A is defined as $A+A^2+\ldots$ and the reflexive transitive closure of A is defined as $I+A"$ where I is the identity matrix, i.e., $I(i,i) = 1$ and $I(i,j) = 0$ for $i \neq j$ and $1 \leq i,j \leq n$.

An edge is called a self-loop if it begins and ends at the same node. Two edges are said to be parallel if they

also have the same pair of end nodes (in the case of directed graph, if they have the same direction). A graph is simple if it has neither self-loops nor parallel edges. Here and hereafter, the graphs under consideration are simple, and the nodes in an n-node graph are labelled with the integers 1 through n, i.e., $V = \{1,2,\ldots,n\}$.

In a graph, a path of length n-1 with endpoints v and w is a sequence of nodes $v = y(1),y(2),\ldots,y(n) = w$ such that $(y(i-1),y(i))$ is an edge for $1 \leq i \leq n$. A path is simple if $y(1),y(2),\ldots,y(n)$ are distinct nodes. A cycle is a simple path from v to v containing at least two edges for directed graph and three edges for undirected graph. A graph which contains no cycle is called acyclic.

G is connected if for every pair of distinct nodes $v,w \in V$, there is a path from v to w (i.e., w is said to be reachable from v) and from w to v. A connected component of G is a maximal connected subgraph of G, i.e., it is not a subgraph of any other connected subgraph of G. D is strongly connected if every two nodes are mutually reachable; it is unilaterally connected if for any two nodes, at least one is reachable from the other; and it is weakly connected if every two nodes are joined by a path in which the direction of each edge is ignored. A strongly connected component of D is a maximal strongly connected subgraph; a unilaterally connected component is a maximal unilaterally connected subgraph; and a weakly connected component is a maximal weakly connected subgraph. Thus, D is strongly, unilaterally

or weakly connected if and only if it has exactly one corresponding connected component. D is <u>disconnected</u> if it is not even weakly connected.

An undirected acyclic graph is called a <u>forest</u>. A connected forest is called a <u>tree</u> which has a distinguished node, called a <u>root</u> such that every node in the tree is reachable from the root. A <u>terminally</u> (or <u>initially</u>) <u>rooted tree</u> is an connected directed acyclic graph such that the root has no leaving (or entering) edges and other nodes have exactly one leaving (or entering) edge. A <u>spanning tree</u> of G is an undirected tree that connects all nodes in V. In a forest, if there is a path from v to w, then v is an <u>ancestor</u> of w and w is a <u>descendant</u> of v. Furthermore, if $(v,w) \in E'$, then v is called an <u>immediate ancestor</u> of w and w is an <u>immediate descendant</u> of v.

Throughout this thesis, unless specified otherwise, $\lceil x \rceil$ denotes the smallest integer $\geq x$ (<u>ceiling</u>), $\lfloor x \rfloor$ denotes the greatest integer $\leq x$ (<u>floor</u>), $|V|$ denotes the cardinality (or size) of any set V and log n denotes $\lceil \log_2 n \rceil$.

1.2.2 <u>Representation of Graphs</u>

In a computer, the graph must be represented in a discrete way. A variety of data structures have been developed for this with respect to conventional sequential computers. The convenience of implementation, as well as the efficiency of a graph algorithm, depends on the proper selection of the representation of the graph. Among all the

known data structures for representing a graph G = (V,E) where |V| = n and |E| = m, the adjacency matrix is particularly appropriate to be adopted for unbounded parallel computation. This is mainly because of its natural structure which provides a large amount of parallelism in matrix manipulations and hence the fundamental operations on sets can be done efficiently by simply taking advantage of the matrix indices. For example, INSERT edge, DELETE edge, UPDATE, UNION, FIND and NUMBER can be done in one time unit, while MINIMUN (finding the minimum element of the set) can be done in O(log n) time units which is indeed optimal. Moreover, initialization and reorganization of an adjacency matrix require O(1) and O(log n) time units respectively. Furthermore, the adjacency matrix is relatively easy to implement and can be converted to other representations of the graph, such as adjacency lists, or trees, in O(log n) time units. In contrast, converting adjacency lists to an adjacency matrix takes linear time.

The only weakness of the adjacency matrix representation is its storage requirements which are always proportional to the square of the number of nodes in the graph regardless the number of edges in the graph. For dense graphs, $|E| = O(n^2)$, the difference of storage requirement between adjacency matrix and other graph representations is insignificant. For sparse graphs, |E| = O(n), the difference increases to O(n).

1.3 Previous Results

With the advent of parallel computers, most researches for developing and analysing parallel algorithms have been concentrated on numerical applications. Hence, several excellent surveys of numerical parallel algorithms have also appeared. Miranker [29] summarized the early work in the late sixties, recently Heller [20] has presented a more complete and up-to-date collection of parallel algorithms in numerical linear algebra, Ortega and Voigt [32] gave a detailed account of algorithms for solving differential equations on vector computers, and Sameh and Kuck [36] described direct parallel algorithms for solving systems of linear equations to a greater depth.

Along the growth of development of parallel algorithms, nevertheless, graph problems have received little attention. The best known upper time and processor bounds of the existing parallel graph algorithms, which are bounded by a polynomial, are displayed in Table 1. Since matrix multiplication and sorting are the useful tools in formulating parallel graph algorithms, their current best upper time and processor bounds are also included in Table 1.

Problem	Time	Processor	Ref.
Sort n Elements	$O(\log n)$	$n \log n$	[34]
Find the Minimum Element of a Set of n Elements	$O(\log n)$	$\lceil n/\log n \rceil$	[37]
Multiply 2 nxn Matrices	$O(\log n)$	$n^{2.81}/\log n$	[10]
Find the Transitive Closure of an Undirected Graph G	$O(\log^2 n)$	$n \lceil n/\log n \rceil$	[37]
Find the Transitive Closure of a Directed Graph D	$O(\log^2 n)$	$n^{2.81}/\log n$	[10]
Find All-Pairs Shortest Path	$O(\log^2 n)$	$n^2 \lceil n/\log n \rceil$	[37]
Find an Absolute Sink	$O(\log n)$	n^2	[19]
Verify Bipartite Graph	$O(\log^2 n)$	n^3	[19]
Find a Spanning Tree of G	$O(\log^2 n)$	$n \lceil n/\log n \rceil$	[37]
Find the Minimum Spanning Trees of G	$O(\log^2 n)$	$n \lceil n/\log n \rceil$	[37]
Find the Connected Components of G	$O(\log^2 n)$	$n \lceil n/\log n \rceil$	[37]
Find the Biconnected Components of G	$O(\log^2 n)$	$n^2 \lceil n/\log n \rceil$	[37]
Find the Bridge Connected Components of G	$O(\log^2 n)$	$n^2 \lceil \log n \rceil$	[37]
Find the Weakly Connected Components of D	$O(\log^2 n)$	n^3	[2]
Find the Strongly Connected Components of D	$O(\log^2 n)$	n^3	[2]
Find the Dominators of D	$O(\log^2 n)$	$\log n(n^{3.81})$	[37]
Find a Cycle of G	$O(\log^2 n)$	n^2	[37]
Find a Cycle Basis of G	$O(\log^2 n)$	n^3	[37]
Find the Shortest Cycle of D	$O(\log^2 n)$	$n^2 \lceil n/\log n \rceil$	[37]

Table 1: Summary of previous results on graph problems

CHAPTER 2

LOWER BOUNDS AND BASIC ALGORITHMS

2.1 Lower Bounds

As an immediate consequence of our model of unbounded parallel computation, the maximum parallelism is achievable when the steps of a computation are completely independent, i.e., all operands are available at the same time. Since each processor uses only unary or binary operations, a simple lower bound on the computation time can be concluded below.

Lemma 2.1: Computing a function in n steps on a single processor can be accomplished in one step on n processors if and only if the operands of each step are independent.

Proof : The proof is trivial. □

The independency of operands, however, may not always exist. One example is to compute $A(n) = a(1) \otimes a(2) \otimes \ldots \otimes a(n)$, where $\otimes$ is any associative binary operation. This problem is the simplest instance of combining n inputs into one output. A(n) can be computed in log n time units with $\lfloor n/2 \rfloor$ processors by using recursive doubling, i.e., repeatedly separate each computation into two independent parts of equal size which are then computed in parallel. This method is actually a parallel version of divide-and-conquer which

can be represented by a binary tree, called binary computation tree, in which the root corresponds to the result, the terminal nodes correspond to the input operands, the internal nodes correspond to the operations and the height (depth) of the tree corresponds to the number of parallel operations performed.

The following facts are immediate from a binary tree.

Lemma 2.2: A binary tree with depth i - 1 has at most $2^i - 1$ nodes.

Lemma 2.3: A binary tree with n nodes has depth at least log n.

These facts lead to the following well-known lower bound for computing A(n) with unbounded parallelism which has been referred to as the fan-in argument. It was generalized to the case with K processors available by Munro and Paterson [30].

Lemma 2.4 (fan-in argument): At least $\lceil \log n \rceil$ parallel operations are required to compute a result which depends on n arguments.

Theorem 2.5 : Suppose the computation of a single element Q requires $q \geq 1$ binary arithmetic operations. Then the shortest computation of Q with K processors is at least

$r(q+1-2^i)/K_1+i$ time units if $q \geq 2^i$ and $\log(q+1)$ otherwise, where $i = \log K$.

Despite its simplicity, Theorem 2.5 is very important because it allows the translation of complexity theorems from sequential computations to parallel computations. Based on the fan-in argument, Arjomandi [2] showed that the lower time bound on verifying connectivity in an n-node undirected graph G is $\log(n-4)-1$ while the lower time bounds on verifying strong, unilateral and weak connectivity in an n-node directed graph D are $\log(n-3)-2$, $\log(n-5)-1$ and $\log(n-4)-1$ respectively. Also, Savage [37 p.115-120] proved that the lower time bounds on determining biconnectivity, bridge connectivity, minimum spanning trees and a cycle in G, and a dominator and a cycle in D are all $2\log n$ with at most a constant difference.

Intuitively, in order to determine various graph properties, in the worst case, all entries in the corresponding adjacency matrix of the graph (i.e. all edges in the graph) have to be examined. For many classes of graph properties, various researchers, namely Best et al. [8], Holt and Reingold [22], Kirkpatrick [24], and Rivest and Vuillemin [35] have shown sequential lower time bounds of at least $O(n^2)$. Hence, the parallel lower time bound of $\log n^2 = 2\log n$ follows immediately from the fan-in argument which gives a good estimate.

2.2 Basic Algorithms

Computation of $A(n) = a(1) \otimes a(2) \otimes \ldots \otimes a(n)$ and matrix multiplication have been the bottlenecks in solving most of the graph problems, where $\otimes$ is any associative binary operation. Therefore reductions of time and processor requirements on these two bottlenecks will substantially reduce the overall time and processor requirements. In Section 2.2.1, we present two optimal algorithms for computing A(n). In Section 2.2.2, previous results on matrix multiplication and its applications are discussed. Moreover, we will point out that the processor bound of many existing graph algorithms, which are based on matrix multiplication, can be reduced simply by using the current best matrix multiplication algorithm.

2.2.1 Computation of A(n)

In the preceding section, we have shown that A(n) is computable in log n time units with at most $\lfloor n/2 \rfloor$ processors using recursive doubling. By inspection, the maximum number of processors merely occurs at the first level of the binary computation tree. In the subsequent levels, half of the number of processors working at any level are idle at the next level. As a result, at the last level, only one processor is working. Clearly, if $\lfloor n/4 \rfloor$ processors are available, the computation time of A(n) only increases one time unit (i.e. first level takes 2 time units instead of

one time unit). This indicates that it is not always desirable to reduce the computation time to an absolute minimum. Having in mind our second objective, namely processor optimization, it is natural to ask:

(A) whether wiser utilization of processors can be organized so that the processor requirement can be minimized without affecting the $O(\log n)$ time bound significantly?

In practice, a related question, which has received much attention, is addressed as follows:

(B) Given P processors, at most how many time units are needed to compute A(n)?

An answer to (A) can be obtained by interpolating different P into (B). Thus, we concentrate our attention on tackling (B).

For a problem of size n, suppose there exists an algorithm using t time units, P processors and q operations. In order to transform the existing algorithm into one that requires a smaller number of K processors, two principles, namely algorithm decomposition and problem decomposition due to Hyafil and Kung [23], are introduced. The idea of algorithm decomposition is to decompose each step i of the existing algorithm into $\lceil q(i)/K \rceil$ substeps so that each of them can be done in one time unit with K processors, where q(i) is the number of operations in step i. Hence, the total time T is bounded as follows [9]:

$$T = \sum_{i=1}^{t} \lceil q(i)/K \rceil \leq t + (q-t)/K \qquad \text{where } \sum_{i=1}^{t} q(i) = q$$

The idea of problem decomposition is to partition the problem into subproblems so that each of them can be solved by the existing algorithm with K processors. Based on this idea, Savage [37 p.17] developed an algorithm for finding the minimum of n elements in less than 2log n time with $\lceil n/\log n \rceil$ processors, which is, in fact, the best answer to (A).

To answer (B), we construct two different algorithms below which basically simulate the binary computation tree based on the above stated principles. The first, named Algorithm A(n)1, incorporates algorithm decomposition with a circular queue which is used to store the operands. It is a variant of Heller's associative fan-in algorithm [20]. Assume that input operands are stored in a circular queue of size n and K processors are available.

Algorithm A(n)1

1. **while** more than one operand in the circular queue do
2. Fetch two operands from the circular queue to each processor and fill up as many processors as possible, execute the specified ⊗ operation, and then store the results to the end of the circular queue.

Under the assumption of our model of parallel computation, step 2 can be done in one time unit. The total

time requirement T for computing A(n) is equal to the number of iterations of step 2. At each iteration, K operands are reduced. After $\lfloor n/K \rfloor - 1$ iterations, $K\lfloor n/K \rfloor - K$ operands are reduced and only $r = n + K - K\lfloor n/K \rfloor$ operands remain which can be done in another $\log(n + r)$ time units to produce the final result. Thus,

If $K < \lfloor n/2 \rfloor$, $T = \lfloor n/K \rfloor - 1 + \log(K+n-K\lfloor n/K \rfloor)$

If $K \geq \lfloor n/2 \rfloor$, $T = \log n$

In order to show these time bounds are indeed equivalent to the lower time bounds shown in Theorem 2.5, we give the following lemma.

Lemma 2.6: T1 = T2 for all n where

$$T1 = \begin{cases} \lceil (n-2^i)/K \rceil + i & \text{if } n-1 \geq 2^i \\ \log n & \text{otherwise} \end{cases}$$

$$T2 = \begin{cases} \lfloor n/K \rfloor - 1 + \log(K+r) & \text{if } \lfloor n/2 \rfloor > K \\ \log n & \text{otherwise} \end{cases}$$

$i = \log K$ and $0 \leq r = n-K\lfloor n/K \rfloor < K$.

Proof: Since $2^i \geq K > 2^i/2$

if $n-1 < 2^i$

$\Longrightarrow$ $2K > 2^i > n-1$ $\Longrightarrow$ $K \geq \lfloor n/2 \rfloor$

Therefore there are only three major cases to consider based on the range differences.

Case 1: $n-1 < 2^i$ and $\lfloor n/2 \rfloor \leq K$

$T1 = T2 = \log n$

Case 2: $n-1 \geq 2^i$ and $\lfloor n/2 \rfloor \leq K$

$\Longrightarrow$ $2x2^i \geq 2K \geq 2\lfloor n/2 \rfloor \geq 2^i$

$\Longrightarrow \quad i+1 = \log K + 1 = \log n$ and $K \geq n-2^i \geq 1$

Hence $T1 = \lceil (n-2^i)/K \rceil + i = 1 + \log n - 1$

$= \log n$

$= T2$

Case 3: $n-1 \geq 2^i$, $\lfloor n/2 \rfloor > K$ and $0 \leq r = n-K\lfloor n/K \rfloor < K$

There are two minor cases to consider based on whether K is power of 2. Two subcases arise when considering whether n is a multiple of K.

Case 3.1: $K = 2^i$

Case 3.1a: $n/K = t$ = integer

$\Longrightarrow \quad \lceil n/K \rceil = \lfloor n/K \rfloor = t$ and $r = 0$

Hence $T1 = \lceil (n-K)/K \rceil + i = t - 1 + i$

and $T2 = \lfloor n/K \rfloor - 1 + \log(K) = t - 1 + i$

Case 3.1b: $n/K \neq t$

$\Longrightarrow \quad \lceil n/K \rceil = \lfloor n/K \rfloor + 1$ and $K > r > 0$

$\Longrightarrow \quad \log 2K = \log(K+r) = 1 + \log K = 1 + i$

Hence $T1 = \lceil (n-K)/K \rceil + i = \lceil n/K \rceil - 1 + i = \lfloor n/K \rfloor + i$

and $T2 = \lfloor n/K \rfloor - 1 + \log(K+r) = \lfloor n/K \rfloor + i$

Case 3.2: $2^i > K > 2^i/2 \Longrightarrow 2^i-K < 2^i/2 < K$

Case 3.2a: $n/K = t$

$\Longrightarrow \quad \lceil n/K \rceil = \lfloor n/K \rfloor = t$ and $r = 0$

Hence $T1 = \lceil (n-2^i)/K \rceil + i$

$= \lceil (n-K)/K \rceil + \lceil (K-2^i)/K \rceil + i = t - 1 + i$

and $T2 = \lfloor n/K \rfloor - 1 + \log(K) = t - 1 + i$

Case3.2b: $n/K \neq t$

$\Longrightarrow \quad \lceil n/K \rceil = (n-r)/K + 1 = \lfloor n/K \rfloor + 1$ and $K > r > 0$

Since $2x2^i > 2K > K+r \Longrightarrow i+1 > \log(K+r)$

if $K+r > 2^i$

$\Longrightarrow$ $K+r-2^i < K$ and $\log(K+r) = i+1$

then $T1 = \lceil (n-K-r)/K \rceil + \lceil (K+r-2^i)/K \rceil + i = \lfloor n/K \rfloor + i$

$T2 = \lfloor n/K \rfloor - 1 + \log(K+r) = \lfloor n/K \rfloor + i$

if $K+r \leq 2^i$

$\Longrightarrow$ $2^i-K-r < K$ and $\log(K+r) = i$

then $T1 = \lceil (n-K-r)/K \rceil + \lceil (K+r-2^i)/K \rceil + i = \lfloor n/K \rfloor - 1 + i$

$T2 = \lfloor n/K \rfloor - 1 + \log(K+r) = \lfloor n/K \rfloor - 1 + i$

Having shown $T1 = T2$ in all cases, the proof is completed.

□

Hence, in both cases, the exact minimum number of time units are achieved and thus the time bounds of Algorithm A(n)1 are tight.

The second, named Algorithm A(n)2, is a generalized version of Savage's partition method for finding the minimum of n elements [37 p.17].

Algorithm A(n)2

1. if $K \geq \lfloor n/2 \rfloor$ then use recursive doubling to compute A(n).
2. Partition the problem, A(n), into K subproblems, S(1),S(2),...,S(K), of $\lfloor n/K \rfloor$ operands each and $0 \leq r = n-K\lfloor n/K \rfloor < K$ operands remain. For example, $S(1) = a(1) \circledast a(2) \circledast \ldots \circledast a(\lfloor n/K \rfloor)$. Assign one processor to each S(i) and then compute all S(i) in parallel.
3. Use recursive doubling tc compute A(n) with all S(i) and the r remaining operands as input.

If $K \geq \lfloor n/2 \rfloor$, clearly step 1 can be done in log n time units by using recursive doubling with at most $\lfloor n/2 \rfloor$ processors. If $K \leq \lfloor n/2 \rfloor$ and $0 \leq r < K$, in step 2, each of the K processors compute S(i) sequentially in at most $\lfloor n/K \rfloor - 1$ time units, whereas step 3 can be done in log(K+r) time units with at most $K-1 \geq (K+r)/2$ processors by using recursive doubling. Hence the total time requirement is $\lfloor n/K \rfloor - 1 + \log(K+r)$. In both cases, the same time bounds as Algorithm A(n)1 are achieved. Thus we have established the following theorem.

Theorem 2.7: Given K processors, Algorithm A(n)1 or A(n)2 computes A(n) in $\lfloor n/K \rfloor - 1 + \log(K+r)$ time units if $\lfloor n/2 \rfloor > K$ and log n time units if $\lfloor n/2 \rfloor \leq K$, where $r = n - K\lfloor n/K \rfloor$. Moreover, the time bounds in both cases are tight.

By substituting $K = \lceil n/\log n \rceil$ and $\oplus$ = Min into Theorem 2.7, we obtain $T(P) = 2\log n - \log\log n - 1$ with $S(P) = (n-1)/(2\log n - \log\log n - 1)$ and $E(P) = O(1)$, which is basically Savage's result. By interpolating different K's into Theorem 2.7, one can easily verify that $K = \lceil n/\log n \rceil$ is minimal subject to the constraint of O(log n) time units. Therefore it is the best answer to (A). Also, this result can be directly applied to reduce the processor requirement of Goldschlager's Algorithm for finding the absolute sink

(i.e., node has in-degree n - 1 and out-degree 0) [19] from n^2 to $n\lceil n/\log n\rceil$ with the same time requirement (i.e., $O(\log n)$).

2.2.2 Matrix Multiplication

Multiplying 2 nxn matrices in our model of unbounded parallel computation by the "high school" method, which can be regarded as computing n^2 vector inner products in parallel, takes $O(\log n)$ time units with n^3 processors if simultaneous read is allowed. Combining Algorithm A(n)1 or A(n)2 with the high school method, the processor bound can be lowered to $n^2\lceil n/\log n\rceil$. Csanky [13] developed the first parallel version of Strassen's matrix multiplication algorithm [39] which runs in $O(\log n)$ time units with $n^{2.81}$ processors. Recently, a more general parallel version of Strassen's algorithm, which runs in $O(n^{2.81}/P)$ time units with $P \leq n^{2.81}/\log n$ processors, $S(P) = P$ and $E(P) = O(1)$, has been constructed by Chandra [10]. A practical advantage of Chandra's algorithm is free of memory conflict.

Furman [17] and Munro [31] independently observed that the transitive closure of an nxn Boolean matrix A can be accomplished by matrix multiplications with the two operations x and + of inner product being changed to logical AND and OR correspondingly. Moreover, Munro [31] presented an $O(n^{2.81})$ sequential algorithm which essentially requires one matrix multiplication. By squaring A log n times, the

transitive closure of A can be computed in $O(\log^2 n)$ time units with n^3 processors using the high school method [21], and in $O(n^{2.81}(\log n)/p)$ time units with $P \leq n^{2.81}/\log n$ processors, $S(P) = O(P)$ and $E(P) = O(1)$ using Chandra's parallel Strassen's algorithm [10]. The speedup and its efficiency over the best sequential algorithm are $S(P) = O(P/\log n)$ and $E(P) = O(1/\log n)$. Employing high school method to compute transitive closure, Goldschlager [19] verified bipartiteness and Arjomandi [2] determined the connected components in $O(\log^2 n)$ time units with n^3 processors. The processor requirements for both problems can be decreased to $n^{2.81}/\log n$ if Chandra's algorithm is used. Analogously, the $\log n(n^{3.81})$ processor requirement of Savage's $O(\log^2 n)$ algorithm for finding dominators [37 p.70-71] can be improved to $n^{3.81}/\log n$ by using Chandra's algorithm.

If A is a symmetric matrix, the problem of finding the transitive closure reduces to identifying its connected components and was efficiently solved by Savage [37 p.50] in $O(\log^2 n)$ time units with $\lceil n/\log n \rceil$ processors. The processor requirement of her method, indeed, can be further reduced to $\lceil n/\log^2 n \rceil$. We will discuss the details in Section 3.1.4.

By replacing + and Min with x and + in performing inner product denoted $\oplus$, Backhouse and Carre [4] extended the matrix multip method to find the all-pairs shortest path of G if no negative cycles are present. Savage [37 p.104-109]

presented an $O(\log^2 n)$ algorithm for this problem with $n^2\lceil n/\log n\rceil$ processors. She then used it to develop an $O(\log^2 n)$ algorithm for determining the shortest cycle in an n-node directed graph using $n^2\lceil n/\log n\rceil$ processors. However, if Chandra's algorithm is employed, the processor requirements of the above two problems can both be lowered to $n^{2.81}/\log n$.

Furthermore, Lawler [26 p.91] pointed out that detecting the existence of negative cycles or finding the length of the shortest cycle become a by-product of the matrix multiplication method. For detecting the existence of negative cycles in an n-node weighted graph G: after squaring the weight matrix W of G log n times, if the minimum of the diagonal elements of the resultant weight matrix is negative, negative cycle exists. The parallel realization, Algorithm NEG.CYCLE, is presented below.

Algorithm NEG.CYCLE

comment: find the transitive closure of the weight matrix W by Chandra's algorithm.

1. for i=1 until log n do W <— Min{W,W⊗W}
2. if Min{W(i,i)|∀i}<0 then return 'Negative cycle'
   else return 'No negative cycle'

Step 1 of Algorithm NEG.CYCLE can be computed in $O(\log n(n^{2.81})/P)$ time units with $P \leq n^{2.81}/\log n$ processors by using Chandra's algorithm, while step 2 can be determined

in log n + 1 time units with at most $\lfloor n/2 \rfloor$ processors. Hence, the total time and processor bounds are $O(\log n(n^{2.81})/P)$ and $P \leq n^{2.81}/\log n$ respectively.

For finding the length of the shortest cycle: after squaring the adjacency matrix log n times, the smallest value of the diagonal elements of the resultant adjacency matrix is the length of the shortest cycle. The same time and processor bounds as detecting negative cycle are obtained.

CHAPTER 3

SOME GRAPH CONNECTIVITY PROBLEMS

In this chapter, we consider problems dealing with the connectivities in graphs. It has been shown in Section 2.1 and Table 1 that the lower and upper time bounds on verifying different connectivities in graphs are $O(\log n)$ and $O(\log^2 n)$ respectively. If a graph is disconnected, it is desirable to identify all the corresponding connected components. By definition, a graph G is connected if and only if there exists exactly one connected component in G. Whence, verifying connectivity in G becomes a consequence of finding the corresponding connected components. Besides Algorithm UNI.CON and Algorithm ACYCLIC, which are merely for verifying unilateral connectivity and acyclicness, algorithms presented in this chapter are mainly focused on the problems of finding different connected components. Problems in undirected and directed graphs are discussed separately in Sections 3.1 and 3.2.

3.1 Finding Connected Components of Undirected Graphs

Based on high school method to compute the reflexive transitive closure of an nxn matrix in $O(\log^2 n)$ time units on n^3 processors, Arjomandi [2] solved the problem of finding the connected components of an n-node undirected graph in $O(\log^2 n)$ time units on n^3 processors which can be lowered to $n^{2.81}/\log n$ processors by using Chandra's parallel Strassen's algorithm. Later, Hirschberg [21] reduced the processor requirement to n^2 while maintaining the $O(\log^2 n)$ time units. Recently, Savage [37 p.49] has improved the processor bound of Hirschberg's method to $\lceil n/\log n \rceil$. We are going to show that this processor bound, in fact, can even further be improved to $\lceil n/\log^2 n \rceil$ with linear speedup and efficiency $O(1)$. Hirschberg's algorithm is firstly described in Section 3.1.1 and the corrections of his algorithm are discussed in Section 3.1.2. The improved algorithm is presented in Section 3.1.3 and its applications are mentioned in Section 3.1.4.

3.1.1 General Hirschberg's Algorithm

In general, Hirschberg's Algorithm CONNECT (see Figure 1) for finding connected components of an undirected graph G can be interpreted as follows:

Step 1. (Uniform Smallest Incident Node Selection): For each node i, select the "minimum" edge (i,j) where node j

```
ALGORITHM CONNECT

begin  comment:nxn array A is the input adjacency matrix.
       Connected components are labelled by the smallest
       node label in themselves. Vector D of size n
       stores the output s.t. D(x) indicates to which
       connected component node x belongs.

1.  ∀x do
      begin D(x) <-- x
            C(x) <-- Min{y≠x| A(x,y)=1}
            Flag(x) <-- 1
      end

    for i = 1 until log n do
2.    begin ∀x s.t. Flag(x)=0 do
          begin D(x) <-- Min{D(x),D[C(x)]}
                for j = 1 until log n do
                begin C(x) <-- C[C(x)]
                      D(x) <-- D[C(x)]
                end
          end

3.        ∀x s.t. Flag(x)=0 do D(x) <-- D[D(x)]

4.        ∀x do
          begin A[x,D(x)] <-- 1
                A[D(x),x] <-- 1
          end

5.        ∀x do if D(x)≠x then Flag(x) <-- 0

6.        ∀x s.t. Flag(x)=0 do
          C(x) <-- Min{D(y)≠D(x)|A(x,y)=1} if none then D(x)

7.        ∀x s.t. Flag(x)=1 do
          C(x) <-- Min{C(y)≠D(x)|A(x,y)=1} if none then D(x)
        end
    end
```

Figure 1: Hirschberg's Algorithm CONNECT for Finding the Connected Components

has the smallest node label among all the nodes incident upon node i. The group of nodes that are connected by the selected edges defines a forest. (Isolated nodes, not incident to any edges selected so far, are regarded as trees with one node.)

Step 2. (Path Compression): Identify the trees, i.e., contract each tree to a single node called <u>center</u> which is actually the smallest-labelled node in the tree. Repeat steps 1 and 2 with the new graph defined by the centers as nodes until there is no edge in the new graph.

Step 3. (Clean Up): (This optional step may be executed after any step 2. It simplifies future computations by deleting superfluous edges.) Delete all unselected edges with both endpoints in the same center. For each pair of centers connected by more than one unselected edges, delete all but the "minimum" of such edges connecting the pair of centers.

A <u>tree-loop</u> is a terminally rooted tree which has an additional edge going from the root to one of its ancestors. Thus, in a tree-loop, the number of nodes is equal to the number of directed edges.

In step 1, the Uniform Smallest Incident Node Selection basically defines a spanning forest of at most $\lfloor n/2 \rfloor$ tree-loops each having at least two nodes. Each isolated

node, not incident to any node so far, is a connected component. It will remain isolated in the subsequent processes and may be considered inactive, i.e., no processor is required to perform any further operation for the isolated node. The trick in step 1 is the smallest-labelled incident node selecting rule - a symmetric relation - which assures that the smallest-labelled node (center) of each tree is in the loop and the immediate descendant of the center is also the immediate ancestor of the center. Whence the loop is assured to be of length two. Path compression in step 2 is the crux of this algorithm. At each iteration, the immediate descendants of all the nodes are connected to their immediate ancestors. Hence, the minimum length path i-->j is shorter by a factor of two. In other words, Path Compression can also be viewed as relabelling nodes on the common path with the center label. This technique is used to "bring together" nodes on a common path to the center. Since the longest possible path is of length n-1, at most log (n-1) iterations is required. A new graph is now established with at most $\lfloor n/2 \rfloor$ non-isolated nodes (centers). The process will be repeated until all the centers are determined to be either connected or isolated. Hence, the algorithm correctly finds the connected components in any undirected graph. The foregoing reasoning is summarized in the following sequence of lemmas where details and proofs can be found in [21]:

Lemma 3.1: The smallest-labelled node (center) in a tree-loop, defined by the Uniform Smallest Incident Node Selection, will be in the loop, and the loop will be of size two.

Lemma 3.2: All the nodes in the tree-loop can be relabelled by the center label in at most $\log(n-1)$ iterations.

Lemma 3.3: After each iteration of steps 1 and 2 in the general Hirschberg's algorithm, the number of centers within each connected component will decrease by at least half in those components that have more than one center.

On the unbounded parallel computation model, finding the smallest incident node by recursive doubling and path compression can be computed in $O(\log n)$ time with $n\lfloor n/2 \rfloor$ and n processors respectively. Since the size of the graph (actually the number of non-isolated nodes) is reduced by at least half after each iteration, the process will be repeated at most log n times. Thus the algorithm requires at most $O(\log^2 n)$ time units with $n\lfloor n/2 \rfloor$ processors.

3.1.2 Corrections of Hirschberg's Algorithm

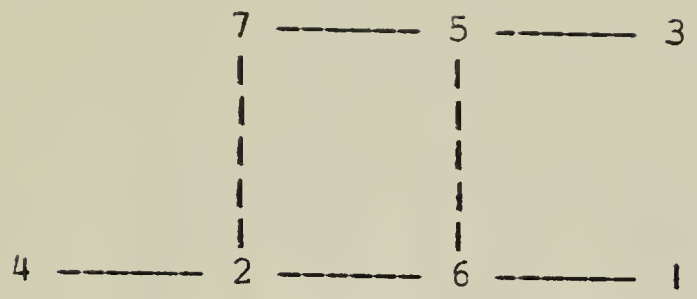
In [21], Hirschberg presented a parallel algorithm for finding the connected components (see Figure 1, Algorithm CONNECT). Three vectors of size n are used to implement Path

Compression: vector Flag indicates the current nodes in the new graph; vector D stores the current label of the nodes; and vector C points to the immediate descendant of each node. Any attempt to follow the algorithm as described may produce incorrect results. The purpose of this section is to point out the mistakes in Hirschberg's algorithm and to offer an alternative which will always produce correct intermediate and final results.

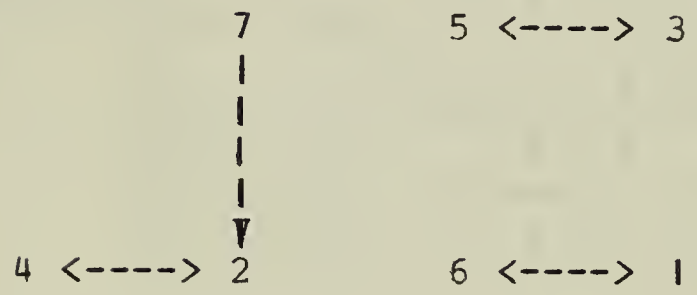
To demonstrate the problem in Hirschberg's Algorithm CONNECT, we consider a simple illustration involving the undirected graph G depicted in figure 2(a). Although the intermediate results from executing lines 4 and 5 of step 2 in Algorithm CONNECT differ depending on whether these lines are executed in series+ or in parallel, the correctness of the whole algorithm is not affected. Therefore we can assume that all instructions at each step will be executed in parallel as long as no write conflict exists++.

+ If Algorithm CONNECT is executed line by line, at the end of the first iteration, the same contradiction - loop of length 3 - is also found. While at the end of the second iteration, node 3 and node 5 are relabelled to 2 instead of the correct value 1 which, however, are relabelled to 1 in the last iteration.

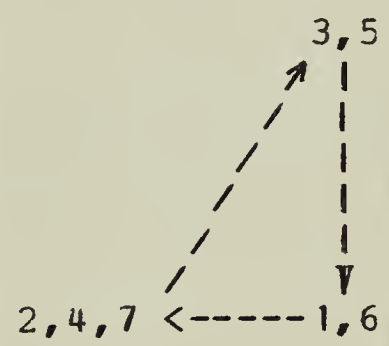
++ Step 4 must be executed in series, otherwise write conflict may occur.



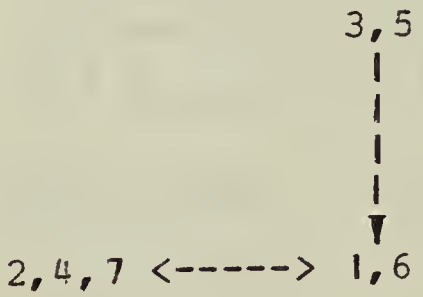
(a) The graph G



(b) Forest of tree-loops obtained after step 1



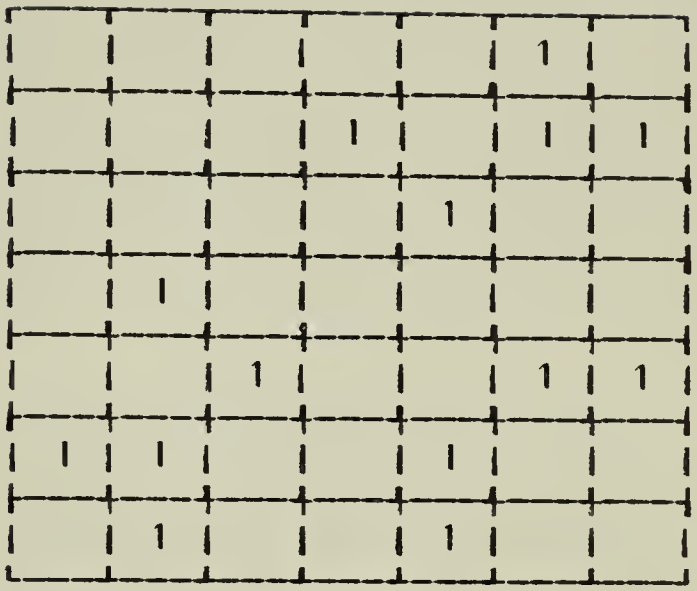
(c) Tree-loop of size 3 obtained after step 2



(d) Correct tree-loop after step 2

Figure 2: Counter Example for Algorithm CONNECT

Adjacency Matrix A of Graph G



Step 1

Flag	1	1	1	1	1	1	1
D	1	2	3	4	5	6	7
C	6	4	5	2	3	1	2

Iteration 1:

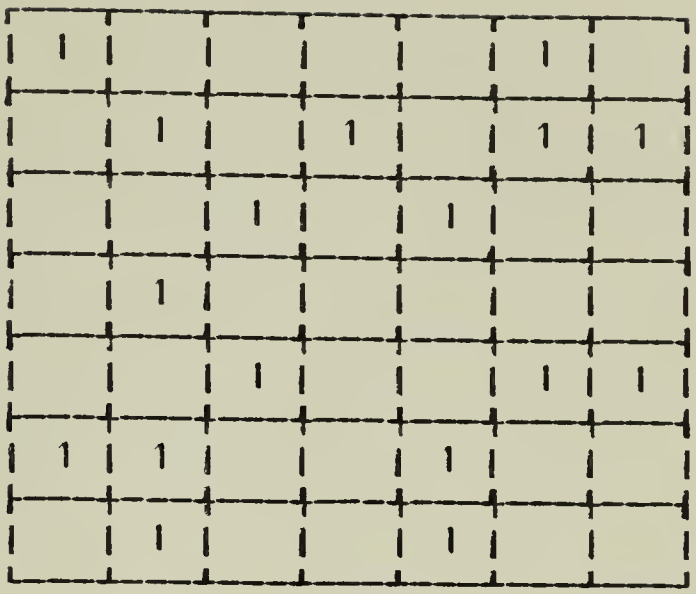
Step 2.1 D	1	2	3	2	3	1	2
Step 2.2(1st) C	1	2	3	4	5	6	4
D	1	2	3	2	3	1	2

Step 2.2(2nd) and step 2.2(3rd) no change

Step 3 no change

Step 4

A



Step 5 Flag

1	1	1	0	0	0	0

Step 6 C

1	2	3	2	1	2	3

Step 7 C

2	3	1	2	1	2	3

Iteration 2:

Step 2.1 D

1	2	1	2	3	1	2

Step 2.2(1st) C

3	1	2	2	1	2	3

D

2	1	1	2	3	1	2

Step 2.2(2nd) C

2	3	1	2	1	2	3

D

1	2	1	2	3	1	2

Step 2.2(3rd) C

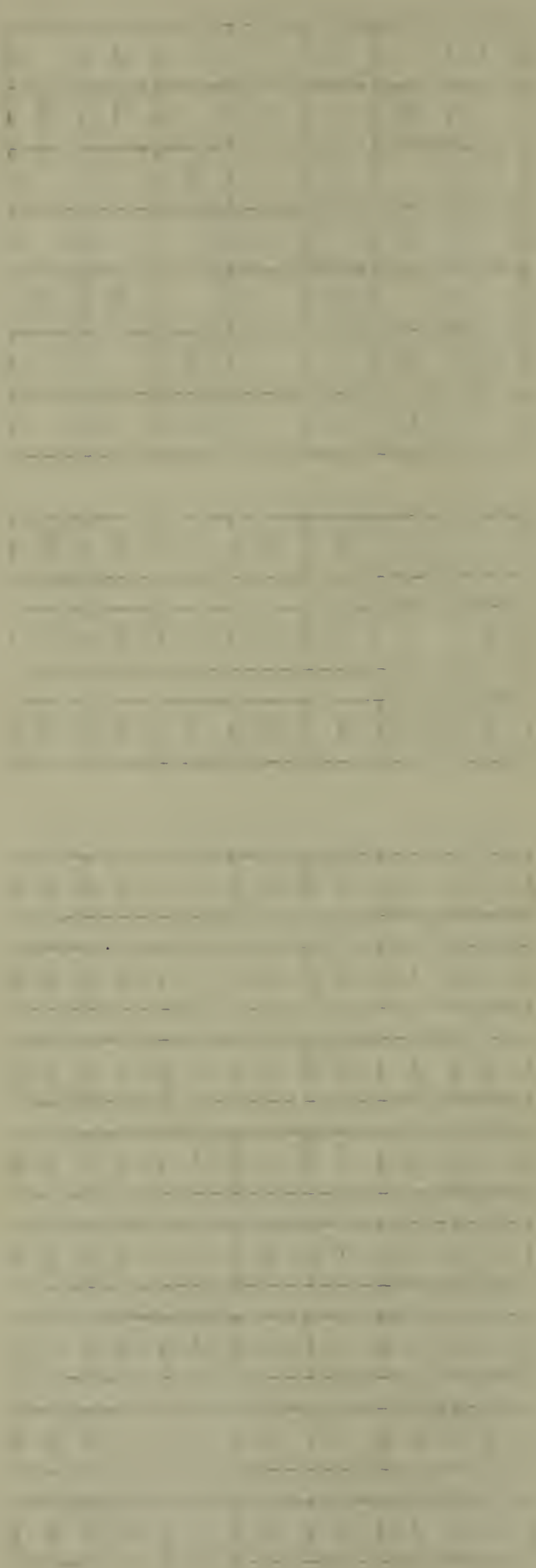
3	1	2	2	1	2	3

D

2	1	1	2	3	1	2

Step 3 D

2	1	1	1	1	2	1



Step 4

A

1	1	1	1	1	1	1
1	1		1		1	1
1		1		1		
1	1					
1		1			1	1
1	1			1		
1	1			1		

Step 5 Flag

0	0	0	0	0	0	0

Step 6 C

1	2	2	2	2	1	2

Step 7 C no change

Iteration 3:

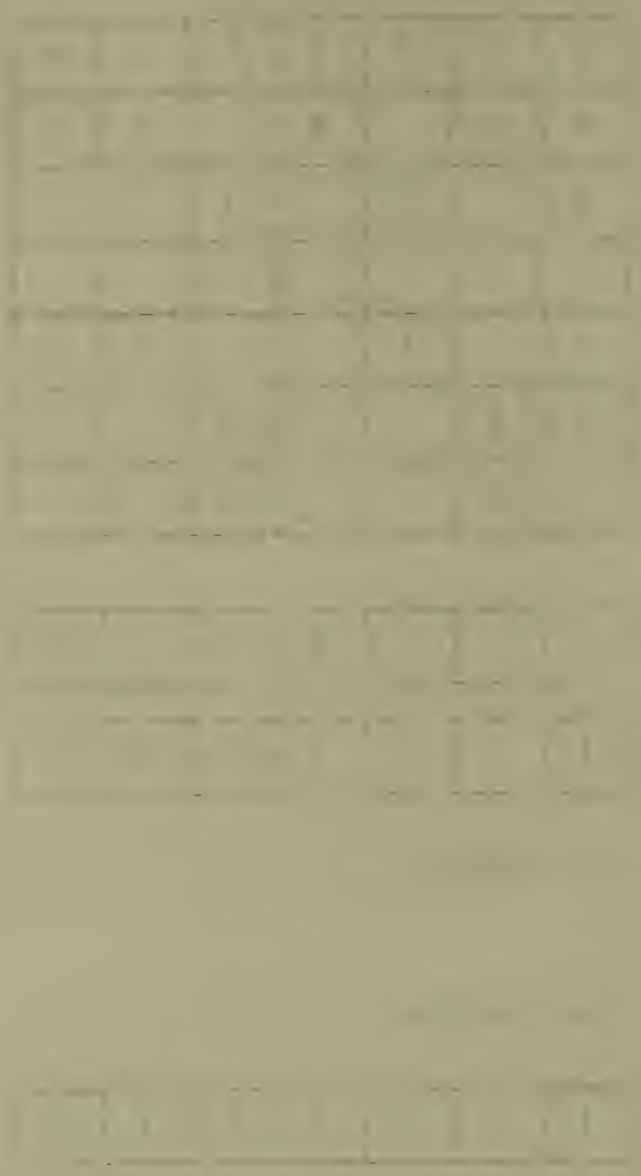
Step 2 no change

Step 3 D

1	2	2	2	2	1	2

Steps 4-7 do not affect the result in the output vector D

At the end of the first iteration, the selected incident node for each center in the new graph G' is stored in vector C which defines a tree-loop as shown in Figure 2(c). Surprisingly, a loop of length three is found which contradicts Lemma 3.1. Consequently, at the end of the second iteration, node 1 and node 6 are relabelled to 2 instead of the correct value 1 while at the end of the last iteration, two connected components are reported in the



output vector D whereas the correct result is one connected component. The correct tree-loop of G' is depicted in Figure 2(d). The essential problem in the algorithm lies in step 7 wherein C(x) is incorrectly found due to the misinterpretation of the unselected edges of the centers. To correct this problem, the following modification of steps 6 and 7 is suggested. (i.e., delete 'Flag(x)=0' from step 6 so that the smallest unselected node directly incident upon each center is included; and replace 'A(x,y)=1' in step 7 by 'D(y)=D(x)'.)

6'. ∀x do

C(x) <-- Min{D(y)≠D(x)|A(x,y)=1} if none then D(x)

7'. ∀x s.t. Flag(x)=1 do

C(x) <-- Min{C(y)≠D(x)|D(y)=D(x)} if none then D(x)

In step 7', nodes belonging to the current centers are identified by checking their current node label with the center label, and consequently step 4 can be eliminated.

A minor error is also found in step 1 when finding the smallest-labelled incident node for each node which is initially isolated. To be consistent with the rest of the algorithm in handling an isolated node, the value of node x is assigned to C(x). Therefore line 3 in step 1 is modified as follows:

C(x) <-- Min{y≠x|A(x,y)=1} if none then D(x)

3.1.3 Modified Hirschberg's Algorithm

In Hirschberg's Algorithm CCNNECT, the n^2 processor requirement is contributed solely by finding the smallest incident node (steps 1, 6 and 7). Therefore reducing the processor requirement in these steps may substantially reduce the overall processor requirement. Finding the smallest incident node is equivalent to finding the minimum of n elements which can be done in $2\log n - \log\log n - 1$ time units with $\lceil n/\log n \rceil$ processors by Theorem 2.7. Based on this result, Savage [37 p.49] reduced the processor requirement of Hirschberg's algorithm to $n\lceil n/\log n \rceil$ with efficiency $O(1/\log n)$.

A question now arises naturally: can the efficiency be reduced to $O(1)$? That is, can the processor requirement be reduced by another factor of log n? The answer to this question turns out to be a favourable one when the optional Clean Up step, stated in general Hirschberg's algorithm, is executed to guarantee the reduction of non-isolated nodes by at least a factor of two after each iteration.

To achieve the desired processor reduction in finding the smallest incident node, the strategy of optimization is applied in two different manners: for the first iteration which has the largest problem size n, minimize the processor requirement $P(n)$ subject to $O(\log^2 n)$ time; for the subsequent iterations, minimize the time subject to $P(n)$ processors. Since the "Min" operation is binary associative, by Theorem 2.7, the smallest incident node can be determined

in less than 2log n time units with $\lceil n/\log n \rceil$ processors. It is obvious that nc more than $2\log^2 n$ time units are required by using $\lceil n/\log^2 n \rceil$ processors. Hence, for n nodes, $P(n) = n\lceil n/\log^2 n \rceil$ processors are required. From the second iteration onwards, the worst case time requirement for each subsequent iteration can also be calculated by Theorem 2.7. As desired, the worst case total time for all log n iterations remains $O(\log^2 n)$. A more detail analysis will be discussed after the modified Hirschberg's algorithm is presented.

Based on the above stated strategy of optimization, Algorithm CONNECT is modified to Algorithm MOD.CONNECT as shown in Figure 3. With respect tc the corrected Algorithm CONNECT in Section 3.1.2, the modifications are discussed as follows:

1. Observe that in Algorithm CONNECT, the number of times to find the smallest incident node is log n + 1 which is one more than the theoretical worst case. Indeed, the last one is virtual work which can be eliminated by slightly rearranging the order of instructions, i.e., move steps 6' and 7' in corrected Algorithm CONNECT up to the beginning of the first for-loop (before step 2) and delete the third line of step 1. (The corresponding steps in Algorithm MOD.CONNECT are steps 2 and 3.)
2. The adjacency matrix is updated by perfcrming logical OR between the selected columns (step 8). Since vector Flag

```
   Algorithm MOD.CONNECT

   begin  comment:Flag(x)=1 indicates node x is a current
          center and column x stores the updated
          information.

   ∀x do comment: Initialization
1    begin D(x) <-- x
           Flag(x) <-- 1
     end
   for i=1 until log n do
     begin comment: Uniform Smallest Incident Node Selection
2          ∀x,y s.t. Flag(y)=1 do
              C(x) <-- Min{D(y)≠D(x)|A(x,y)=1}
                       if none then D(x)
           ∀x s.t. Flag(x)=1 do
3          begin C(x) <-- Min{(C(y)≠D(x)|D(y)=D(x)}
                          if none then D(x)
4               if C(x)=D(x) then Flag(x) <-- 0
                comment: Path Compression
5               D(x) <-- Min{D(x),D[C(x)]}
                for j=1 until log(n-1) do
6               begin C(x) <-- C[C(x)]
                      D(x) <-- D[C(x)]
                end
                end
7          ∀x s.t. Flag(x)=0 do D(x) <-- D[D(x)]
           comment: Clean Up (by column contraction)
8          ∀x,y s.t. y=D(y) do
              ∀z s.t. Flag(z)=1 do
                 A(x,y) <-- OR{A(x,z)|D(z)=D(y)}
9          ∀x do if D(x)≠x then Flag(x) <-- 0
     end
   end
```

Figure 3: Modified Hirschberg's Algorithm

has been designed to indicate the current centers, its values can be used as addresses of columns in the subsequent assignement so that only those columns where the appropriate value of Flag is 1 will be fetched. Thus, for finding the smallest incident node, checking condition 'Flag(y) = 1' in step 2 of Algorithm

MCD.CONNECT will assure the desired problem size reduction (i.e., reduction of number of columns) and consequently achieve the desired time reduction.

3. If the whole connected component has merged to a single center (i.e., the center will be isolated), that center will not be considered in the succeeding iterations by having its flag set to zero at step 4.

In order to prove that these modifications to Algorithm CONNECT produce the desired processor reduction, a few definitions and lemmas are needed.

Let Mi be the smallest element among the k selected elements, {M1,M2,...,Mk}, from a vector V of size n. To "merge" the k selected elements means

$$V(Mi) \leftarrow OP\{V(Mj) \neq 0 \mid 1 \leq j \leq k\}.$$

where OP can be Min or logical OR which depends on using the index or the content of V.

Given a vector of n elements, if at least 2 and at most n elements are "merged" to form a new element, then the size of the vector is reduced to at least one and at most $\lfloor n/2 \rfloor$ accordingly. Such a "merge" is called shrink.

Lemma 3.4: Given K processors, performing a shrink operation on a vector V of size n requires at most $\lceil n/K \rceil - 1 + \log K$ time units if $\lfloor n/2 \rfloor > K$ and log n time units if $\lfloor n/2 \rfloor \leq K$.

Proof: Performing a shrink operation on V can be thought of as partitioning V into m groups,

{g(1),g(2),...,g(m)}, and performing "merge" simultaneously on all m groups. Clearly, the parallel computations to "merge" n elements is a binary computation tree which requires n-1 operations (internal nodes) in total. Let T be the time to shrink V, K be the number of processors available.

Case 1: No partition, i.e., all n elements are "merged" to form a single new element. This can be solved optimally by using Algorithm A(n)1 or A(n)2 in Section 2.2.1. Hence, the time bound follows directly from Theorem 2.7, i.e.

$$T = \begin{cases} \lfloor n/K \rfloor - 1 + \log(K+r) & \text{if } \lfloor n/2 \rfloor > K \\ \log n & \text{if } \lfloor n/2 \rfloor \leq K \end{cases} \qquad (1)$$

where $0 \leq r = n - K\lfloor n/K \rfloor < K$.

Case 2: V is partitioned into m groups. It can be considered as splitting an n-leaf binary computation tree into m smaller binary computation subtrees. Because of the splitting, the independence between subtrees increases the parallelism while the total number of internal nodes is decreased. The total number of operations is the sum of the internal nodes of all m binary computation subtrees, i.e.,

$$|g(1)|-1 + |g(2)|-1 + \ldots + |g(m)|-1 = n-m.$$

Case 2a: $K \geq \lfloor n/2 \rfloor$. Assign $\lfloor |g(i)|/2 \rfloor$ processors to group g(i) and then execute all groups in parallel which takes $\text{Max}\{\log|g(i)| \text{ for } 1 \leq i \leq m\} \leq \log n$ time units. The total number of processors required are as follows.

$$\sum_{i=1}^{m} \lceil |g(i)|/2 \rceil \leq \lceil n/2 \rceil .$$

Case 2b: $K < \lceil n/2 \rceil$. Align the m groups of elements as shown in Figure 4 and partition the elements into K sets, each of $\lceil n/K \rceil$ elements, execpt that the last sets has $n - (K-1)\lceil n/K \rceil$ elements.

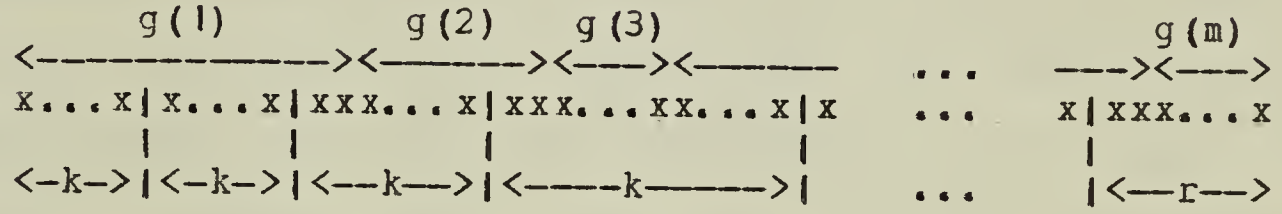


Figure 4: Partition of m Groups into K Sets

Assign one processor to each of the K sets to compute the results in that set. If all the elements in a set belong to the same group, one answer will result from that set. If the elements in a set belongs to several groups, say b groups, then b answers, one for each group, will be obtained. If $b > 2$, at least $b - 2$ answers are final results and at most 2 answers in each set will be combined with answers in other sets to give the final result (the first and the last set have one answer). For example, (see Figure 4), sets 1 and 2 have one answer, set 3 has 2 answers and set 4 has 3 answers while one of them is the final result.

It is obvious that no more than $\lceil n/K \rceil - 1$ time units are needed for computing answers in each set.

Let us assume that n(i) answers will be combined to give the result of g(i). (In Figure 4, n(1) = 3, n(2) = 2 and n(3) = 0.) Assign $\lfloor n(i)/2 \rfloor$ processors to each group to compute the result of that group. Since the sum of all n(i) is less than or equal to 2K - 2, the total number of processors required will be less than K. Each g(i) will take another log n(i) ≤ log K time units to obtain the final result. Thus, the total time requirement is no more than $\lceil n/K \rceil - 1 + \log k$ time units. It is clear that this time requirement is at most one time unit from optimal (i.e., the corresponding case in (1)). □

Theorem 2.7 and Lemma 3.4 give an upper bound on the parallel time complexity for computing a result of n elements and results of groups of n elements. Since the 'Min' operation in steps 2 and 3, and 'OR' operation in step 8 of Algorithm MOD.CONNECT are associative binary operations, Theorem 2.7 and Lemma 3.4 give an upper bound on the total number of time units spent in these steps.

Lemma 3.5: Given K processors, step 2 in Algorithm MOD.CONNECT takes at most T time units where

$$T = \begin{cases} O(\log^2 n) & \text{if } K \geq n\lfloor n/2 \rfloor \\ O(n^2/K + \log n \log K) & \text{if } n \leq K < n\lfloor n/2 \rfloor \\ O(n^2/K) & \text{if } 0 < K < n. \end{cases}$$

Proof: As mentioned earlier (Lemmas 3.1-3.3), further iterations of steps 2-8 merge centers. Step 9 eliminates those merged nodes which are no longer centers by setting their flags to zero. By Lemma 3.3, the number of centers (flagged elements) $|S|$, in each connected component decreases by a factor of at least two after each iteration until the connected component is represented by a single center. Moreover, if the whole connected component has merged to a single center (i.e., the center will be isolated), that center will not be considered in the succeeding iterations by having its flag set to zero at step 4. Thus, we have n flagged elements at the first iteration and have at most $\lfloor n/2^i \rfloor$ flagged elements after i iterations. At step 2, in order to compute all $C(x)$, $p = \lfloor K/n \rfloor$ processors are assigned to each x to compute the minimum value among at most $|S|$ elements. Since 'Min' is an associative binary operation, we can apply Theorem 2.7 to evaluate the time complexity.

Case 1: $K \geq n\lfloor n/2 \rfloor$, we have

$$\sum_{i=0}^{\log n - 1} \log(n/2^i) = O(\log^2 n).$$

Case 2: $n \leq K < n\lfloor n/2 \rfloor$ implies $1 \leq p = \lfloor K/n \rfloor < \lfloor n/2 \rfloor$ processors can be assigned to compute each $C(x)$. Since $|S|$ is reduced by at least half after each iteration, $|S|$ is at

most 2p after $t = \log n - \lfloor \log p \rfloor$ iterations. Thus we have

$$\sum_{i=0}^{t-1} (\lceil \lfloor n/2^i \rfloor / p \rceil - 1 + \log p) + \sum_{i=t}^{\log n-1} \log(n/2^i)$$

$$\leq \lceil 2n/\lfloor K/n \rfloor \rceil + t\log \lfloor K/n \rfloor + (\log \lfloor K/n \rfloor)^2$$

$$\leq O(n^2/K + \log n \log K).$$

Case 3: $0 < K < n$ implies that only K C(x) can be computed in parallel with one processor each. This takes $|S| - 1$ time units. For n C(x), the same computation is repreated no more than $\lceil n/K \rceil$ times. Thus, we have

$$\sum_{i=0}^{\log n-1} (\lfloor n/2^i \rfloor - 1) \lceil n/K \rceil$$

$$\leq 2n \lceil n/K \rceil \leq O(n^2/K). \qquad \square$$

Lemma 3.6: Given K processors, step 3 in algorithm MOD.CCNNECT takes at most T time units where

$$T = \begin{cases} O(\log^2 n) & \text{if } K \geq n\lfloor n/2 \rfloor \\ O(n^2/K + \log^2 n) & \text{if } n \leq K < n\lfloor n/2 \rfloor \\ O(n^2/K) & \text{if } 0 < K < n. \end{cases}$$

Proof: Since the number of flagged elements is reduced by at least half after each iteration, the number of processors assigned to compute C(x) can be double after each iteration. For the first iteration, $p = \lfloor K/n \rfloor$ processors are assigned to compute each C(x). After i iterations, $(2^i)p$ processors can be assigned for each C(x). Thus, we have

Case 1: $K \geq n\lfloor n/2 \rfloor$ implies $p = \lfloor K/n \rfloor \geq \lfloor n/2 \rfloor$ processors can be assigned to compute each C(x).

$$\sum_{i=0}^{\log n-1} \log n = O(\log^2 n).$$

Case 2: $n \le K < n\lfloor n/2 \rfloor$ implies $1 \le p = \lfloor K/n \rfloor < \lfloor n/2 \rfloor$ processors are assigned to compute each C(x) for the first iteration. After $t = \log n - \log p + 1$ iterations, $2^t p \ge \lfloor n/2 \rfloor$ processors can be assigned to each C(x). Thus, we have

$$\sum_{i=0}^{t-i} (\lceil n/(2^i p) \rceil - 1 + \log(2^i p)) + \sum_{i=t}^{\log n-1} \log n$$

$$\le \lceil 2n/\lfloor K/n \rfloor \rceil + \log n \log n + \log^2 n$$

$$\le O(n^2/K + \log^2 n).$$

Case 3: $0 < K < n$ implies only K C(x) can be computed in parallel with one processor for each C(x) which takes $n/2^i - 1$ time units for iteration i. The same computation must be repeated $\lceil (n/2^i)/K \rceil$ time units. Thus, we have

$$\sum_{i=0}^{\log n-1} (n/2^i - 1)\lceil (n/2^i)/K \rceil = 2n\lceil 2n/K \rceil$$

$$\le O(n^2/K).$$ □

Lemma 3.7: Given K processors, step 8 in Algorithm MOD.CONNECT takes at most T time units where

$$T = \begin{cases} O(\log^2 n) & \text{if } K \ge n\lfloor n/2 \rfloor \\ O(n^2/K + \log n \log K) & \text{if } n \le K < n\lfloor n/2 \rfloor \\ O(n^2/K) & \text{if } 0 < K < n. \end{cases}$$

Proof: After sets of centers are merged in steps 5 and 6, the adjacency information among the centers is updated in step 8, i.e. center x and center y are connected by setting

A(x,y) to 1, if there exists an edge joining one of the nodes merged into x to one of the nodes merged into y. In step 8, those columns z in the adjacency matrix A corresponding to those nodes z, which are merged to center y, are 'OR' together to give the new column y. Since 'OR' is an associative binary operation and groups of cclumns are 'OR' together to single columns which correspond to the new-formed centers, Lemma 3.4 can be applied to derive the time bound for step 8. There are n rows in A and these rows of elements are considered in parallel. As what is done for step 2, $p = \lfloor K/n \rfloor$ processors are assigned to each row x to compute A(x,y). During the first iteration, p processors are assigned for each row to deal with n elements; and for each succeeding iteration, the number of elements to be dealt by the p processors is at most half of the number in the previous iteration. Thus, applying the same kind of analysis as in the proof of Lemma 3.5, we derive the same time bound as stated in Lemma 3.5. □

Theorem 3.8: Algorithm MOD.CONNECT finds the connected components of an undirected graph with n nodes in time $O(n^2/K + \log^2 n)$ using K processors.

Proof: The time and processor requirements are listed in Table 2. From Table 2, K processors suffice to determine the connected components of an undirected graph with n nodes in time $O(n^2/K + \log^2 n)$. □

Step	Total Time			Processors		
	case1	case2	case3	case1	case2	case3
1	$O(n/K)$	$O(1)$	$O(1)$	K	n	n
2	$O(n^2/K)$	$O(n^2/K+\log n \log K)$	$O(\log^2 n)$	K	nK	n^2
3	$O(n^2/K)$	$O(n^2/K+\log^2 n)$	$O(\log^2 n)$	K	nK	n^2
4	$O(n/K)$	$O(\log n)$	$O(\log n)$	K	n	n
5	$O(n/K)$	$O(\log n)$	$O(\log n)$	K	n	n
6	$O(n\log n/K)$	$O(\log^2 n)$	$O(\log^2 n)$	K	n	n
7	$O(n/K)$	$O(\log n)$	$O(\log n)$	K	n	n
8	$O(n^2/K)$	$O(n^2/K+\log n \log K)$	$O(\log^2 n)$	K	nK	n^2
9	$O(n/K)$	$O(\log n)$	$O(\log n)$	K	n	n

where case1: $0 < K < n$
case2: $n \leq K < n\lfloor n/2 \rfloor$
case3: $K \geq n\lfloor n/2 \rfloor$

Table 2: Total Time and Processor Requirements for Alogrithm MOD.CONNECT

As a by-product of Theorem 3.8, we have the following result.

Corollary 3.9: Given $\lceil n/\log^2 n \rceil$ processors, Algorithm MOD.CONNECT determines the connected components of an undirected graph with n nodes in time $O(\log^2 n)$.

From Table 2, Algorithm MOD.CONNECT takes $T(1) = O(n^2)$ and $T(p) = O(\log^2 n)$ time units with 1 and $p = \lceil n/\log^2 n \rceil$ processors respectively. Hence, the speedup and the efficiency of Algorithm MOD.CONNECT are $S(p) = O(n^2/\log^2 n)$ and $E(p) = O(1)$. This is the best result that uses the least number of processors to find the connected components of an

undirected graph in time $O(\log^2 n)$. The previous results [37 p.45] needs $n_r n/\log n_1$ processors to achieve the same time bound.

It is of interest to compare the time and processor complexities of Algorithm MOD.CONNECT with the time complexity of the corresponding efficient sequential algorithm. Consider a graph $G = (V,E)$ where $|V| = n$ and $|E| = m$. If G is dense, i.e. $m = O(n^2)$, the best sequential algorithm for this problem requires $T(1) = O(m+n) = O(n^2)$ time [43]. This is because any algorithm will, in the worst case, have to look at all of the edges. Therefore the speedup and its efficiency of Algorithm MOD.CONNECT over the best sequential algorithm also are $O(n^2/\log^2 n)$ and $O(1)$ respectively. On the other hand, if G is sparse, i.e. $m = O(n)$, then $T(1) = O(n)$ and $S(P) = O(n/\log^2 n)$ and $E(P) = O(1/n)$. This implies a lot of waste of the processing power to achieve the speedup of $O(n/\log^2 n)$.

Remarks on Algorithm MOD.CONNECT:

1. On average, it is profitable to detect the earliest termination of Algorithm MCD.CONNECT especially when the graph is dense. One of the stopping criteria is when no center is "merged" in the current iteration. The realization is suggested below:

   Insert step 1.5 'Last <-- n' into step 1 and the following two steps at the end of the first for-loop.

   10 Now <-- Sum{Flag(x) | $1 \le x \le n$}

11 if Last=Now then STOP else Last <— Now

In total, step 1.5 and step 11 require 1 and at most log n time units respectively by using one processor; whereas step 10 requires at most $\log^2 n$ time units by using $\lfloor n/2 \rfloor$ processors.

2. When implementing the Clean Up step, one can contract the rows instead of the columns of the adjacency matrix. It will end up with the same time and processor bounds as Algorithm MCD.CONNECT. Furthermore, rows and columns can be contracted one after the other to gain more time reduction for the next contraction and finding the smallest incident node from the second iteration onwards. However, the tradeoff between spending more time on each Clean Up step and less time on subsequent iterations is insignificant.

3.1.4 Applications of Algorithm MOD.CONNECT

The following paragraphs give brief descriptions of how Algorithm MOD.CONNECT is applied to other related problems, such as finding all spanning trees, finding the minimum spanning tree and the transitive closure of an undirected graph; they are intended simply to show the modification of Algorithm MCD.CONNECT accordingly and hence, merely the complexities of the additional and modified steps are given.

Spanning Trees

Generating all spanning trees of a connected, undirected graph G has played an important role in many parallel graph problems. For instance, in finding the biconnected components of G, the first step is to generate all spanning trees of G [37 p.63-64]. Consider what happens in Algorithm MOD.CONNECT. The algorithm basically generates a spanning tree for each connected component by using uniform depth-first search. However, the edges in each spanning tree are not recorded. This can be efficiently implemented by using three vectors Col, Head and Tail where Head(x), Tail(x) $\in$ V and the set {Head(x),Tail(x)|Flag(x)$\neq$1 for $1 \leq x \leq n$} is the set of edges in all spanning trees. Also the column contraction in Algorithm MOD.CONNECT will destroy the correct column indices. As a result, the selected edges can nct be correctly represented by the matrix indices. To remedy this problem, the adjacency matrix is modified by resetting the connected edges in each row i of A with the corresponding column indices and substituting 'Min' for 'OR' in step 8. The realization of the modified Algorithm MOD.CONNECT, called Algorithm SPAN.TREE, is shown in Figure 5.

Resetting A (step 1e) requires $O(\log^2 n)$ time units with $\lceil n^2/\log^2 n \rceil$ processors; initializing Head and Tail (steps 1c and 1d), and updating Head (step 3d) require $O(1)$ time units with n processors; updating Col and Tail (steps 2b and 3c) require at most $O(\log^2 n)$ time units with $n\lceil n/\log^2 n \rceil$

```
   Algorithm SPAN.TREE

   begin
1  ∀x do comment: Initialization
1a   begin D(x) <-- x
1b         Flag(x) <-- 1
1c         Head(x) <-- 0
1d         Tail(x) <-- 0
1e         if A(x,y)=1 then A(x,y) <-- y
     end
   for i=1 until log n do
     begin comment: Uniform Smallest Incident Node Selection
2          ∀x,y s.t. {D(y)≠D(x) AND Flag(y)=1 AND A(x,y)≠0} do
2a               begin C(x) <-- Min{D(y)} if none then D(x)
2b                     Col(x) <-- Min{y|D(y)=C(x)}
                 end
3          ∀x s.t. Flag(x)=1 do
3a         begin ∀y s.t. {D(y)=D(x) AND C(y)≠D(x)} do
3b               begin C(x) <-- Min{C(y)}
3c                     Tail(x) <-- Min{y|C(y)=C(x)}
3d                     Head(x) <-- A[Tail(x),Col(Tail(x))]
                 end
4                if C(x)=D(x) then Flag(x) <-- 0
                 comment: Path Compression
5                D(x) <-- Min{D(x),D[C(x)]}
6                for j=1 until log(n-1) do
6a               begin C(x) <-- C[C(x)]
6b                     D(x) <-- D[C(x)]
                 end
           end
7          ∀x s.t. Flag(x)=0 do D(x) <-- D[D(x)]
           comment: Clean Up (by column contraction)
8          ∀x,y s.t. y=D(y) do
8a            ∀z s.t. Flag(z)=1 do
8b             A(x,y) <-- Min{A(x,z)≠0|D(z)=D(y)}
9          ∀x do if D(x)≠x then Flag(x) <-- 0
     end
   end
```

Figure 5: Algorithm SPAN.TREE

processors. Hence, the changes do not affect the upper time and processor bounds of Algorithm MOD.CONNECT (Corollary 3.9) and thus Algorithm SPAN.TREE generates all spanning trees of G in $O(\log^2 n)$ time units with $\lceil n/\log^2 n \rceil$ processors.

Minimum Spanning Tree

Given a weighted, connected and undirected graph G, it is often of interest to determine a spanning tree of minimum total edge weight, i.e., such that the sum of the weights of all edges in the tree is minimum. Such a tree is called a minimum spanning tree (MST). In [28], Levitt and Kautz implemented Sollin's algorithm [7 p.189] on their cellular array and yielded the first parallel algorithm for determining MST. By using a transitive closure algorithm to find the spanning trees, Csanky [13] produced an $O(\log^3 n)$ algorithm with $n^{2.81}$ processors. By modifying Algorithm CONNECT to find spanning trees, Savage [37 p.42-45] reduced the time to $O(\log^2 n)$ with $\lceil n/\log n \rceil$ processors. The processor bound of her method can further be improved to $\lceil n/\log^2 n \rceil$ by employing Algorithm SPAN.TREE with a similar modification in selecting smallest incident node. That is, node j is selected to be the smallest incident node for node i if edge (i,j) has the minimum weight among all the edges emanating from node i. Moreover, if there is a tie, then select the edge with smallest node label as before. This minor modification will merely double the time in finding

smallest incident node in Algorithm SPAN.TREE. This is summarized in the corollary below.

Corollary 3.10: The MST of a weighted, connected and undirected graph with n nodes can be determined in $O(\log^2 n)$ time units by using $\lceil n/\log^2 n \rceil$ processors.

Transitive Closure of Undirected Graphs

Once the connected components of an undirected graph G are found, the transitive closure A" of G can be easily obtained with an additional comparison, i.e., A"(i,j) = 1 if and only if i and j belong to the same connected component. This additional comparison can be done in $\log^2 n$ time units by using $\lceil n^2/\log^2 n \rceil$ processors. Hence, with this additional step, Algorithm MCD.CONNECT finds the transitive closure of an undirected graph with n nodes in $O(\log^2 n)$ time units by using $\lceil n/\log^2 n \rceil$ processors. This improves on Savage's $O(\log^2 n)$ algorithm with $\lceil n/\log n \rceil$ processors [37 p.50]. Thus, we have the following corollary.

Corollary 3.11: The transitive Closure of an nxn symmetric Boolean matrix can be determined in $O(\log^2 n)$ time units using $\lceil n/\log^2 n \rceil$ processors.

This method, unfortunately, can not be extended to reduce the processor bound for the problem of finding all-pairs shortest paths.

3.2 Some Connectivity Problems in Directed Graphs

In directed graphs, the properties involving paths, cycles, and connectivity become more complicated than in undirected graphs because of the edge orientations. In this section, we turn our attention to directed graphs and see whether we can get good upper bounds for the problems of finding different connected components.

Weakly Connected Components

The weakly connected components of a directed graph are easily obtained by ignoring the edge directions and removing duplicate edges, and then using Algorithm MOD.CONNECT to find the connected components of the resulting undirected graph. The conversion of the adjacency matrix can be done in $\log^2 n$ time units with $\lceil n^2/\log^2 n \rceil$ processors. Thus, finding weakly connected components of an n-node directed graph takes at most $O(\log^2 n)$ time units with $n\lceil n/\log^2 n \rceil$ processors. The algorithm is as follows:

Algorithm WEAK.CON

1. $\forall i,j$ A(i,j) <-- A(i,j) OR A(j,i)
2. Find the connected components by Algorithm MOD.CONNECT

Strongly Connected Components

Now consider the problem of strongly connected components of a directed graph with n nodes. Based on matrix multiplication for computing the reflexive transitive

closure A" of adjacency matrix A, Arjomandi [2] presented an algorithm for finding the strongly connected components of an n-node directed graph in $O(\log^2 n)$ time units with n^3 processors and $2n^2 + n$ storage. By using Chandra's parallel Strassen's algorithm to compute A", we propose a new Algorithm STRONG.CON below which requires $O(\log n(n^{2.81})/P)$ time units with $P \leq n^{2.81}/\log n$ processors and $n^2 + n$ storage.

Algorithm STRONG.CON

1. Find A", the reflexive transitive closure of A.
2. A"(i,j) <-- A"(i,j) AND A"(j,i)
3. Index(i) <-- the position of the first nonzero entry in row i

In Algorithm STRONG.CON, A requires n^2 storage. Employing Chandra's algorithm to compute A" takes $O(\log n(n^{2.81})/P)$ time units with $P \leq n^{2.81}/\log n$ processors. Since row i of A marks all the reachable pairs of nodes from node i, step 2 only removes the mark of the nonmutually reachable pairs of nodes from A which takes one time unit with n^2 processors. Step 3 uses the smallest-labelled node in each strongly connected components to identify the corresponding components which takes log n time units with n^2 processors and n extra storage. (Notice that due to the symmetry of A", steps 2 and 3 can be performed only on the lower triangular matrix of A" without

affecting the result.) Hence, the total time is $O(\log n(n^{2.81})/P)$ time units with $P \leq n^{2.81}/\log n$ processors. The result is summarized as follows.

Corollary 3.12: Algorithm STRONG.CON determines the strongly connected components of an n-node directed graph in $O(\log n(n^{2.81})/P)$ time units using $P \leq n^{2.81}/\log n$ processors.

Unilateral Connectivity

In the case of finding the unilaterally connected components of an n-node directed graph G, Arjomandi [3] showed that the number of unilaterally connected components can grow exponentially with n. It implies either exponential time or processors are required for any parallel algorithm (otherwise the open question P = NP would be settled.) which is beyond the scope of this thesis. Nevertheless, if the problem is limited to verify unilateral connectivity, we construct a parallel algorithm below which solves this problem in $O(\log n(n^{2.81})/P)$ time units with $P \leq n^{2.81}/\log n$ processors.

Algorithm UNI.CON

1. Find A", the reflexive transitive closure of A.
2. A"(i,j) <— A"(i,j) OR A"(j,i)
3. if A"(i,j)=1 ∀i,j then return 'Unilaterally connected'
   else return 'Non-unilaterally connected'

Step 1 computes all the reachable nodes from all nodes which can be done in $O(\log n(n^{2.81})/P)$ time units with $P \leq n^{2.81}/\log n$ processors using Chandra's algorithm. By definition, if G is unilaterally connected, there must be at least one path between each pair of nodes. This is simulated in steps 2 and 3 using one and $2\log n$ time units, and n^2 and $\lfloor n^2/2 \rfloor$ processors respectively

Hence, Algorithm UNI.CON takes $O(\log n(n^{2.81})/P)$ time units with $P \leq n^{2.81}/\log n$ processors.

Acyclicness

The problem of verifying acyclicness of a directed graph G can be reduced to computing the transitive closure A" of G. If all the diagonal elements of A" are zero, it implies that there exists no cycle in G. The algorithm is as follows.

Algorithm Acyclic

1. Find A", the transitive closure of G
2. if A"(i,i)=0 ∀i then return 'Acyclic'
   else return 'Cyclic'

Step 1 can be done in $O(\log n(n^{2.81})/P)$ time units using $P \leq n^{2.81}/\log n$ processors. Step 2 takes $\log n$ time units with $\lfloor n/2 \rfloor$ processors. Hence, Algorithm Acyclic takes $O(\log n(n^{2.81})/P)$ using $P \leq n^{2.81}/\log n$ processsors.

CHAPTER 4

CONCLUSION

It has been demonstrated that under an idealized model of parallel computation, graph problems can be solved efficiently by representing graphs as adjacency matrices. Two optimal algorithms were presented for computating $A(n) = a(1) \otimes a(2) \otimes \ldots \otimes a(n)$, whose time bounds were proven to be equal to the theoretical lower time bounds in both bounded and unbounded parallelism, where $\otimes$ is any associative binary operation. This result, together with the exploitation of the reducibility cf the problem size by at least half after each iteration, facilitates the reduction of processor requirements by a factor of log n over the existing algorithms for a set of graph problems that can be reduced to the problem of finding the connected components of an n-node undirected graph. Moreover, the number of processors needed to execute each algorithm is optimally utilized (i.e., $E(P) = O(1)$).

In general, the technique developed in Section 3.1.3 for achieving the processor reduction mentioned above can be applied analogously to reduce the processor requirement for any parallel algorithm in which the problem size is reduced by at least half after each iteration. Furthermore, if a problem's size can be reduced by a factor of the square root of n after each iteration, $O(\log^i n)$ time requirement can be

improved to $O(\log^{i-1} n)$ by a similar technique where $i > 1$. An algorithm possessing this property still awaits discovery however.

As pointed out in Section 2.2.2, the processor requirements of many existing parallel graph algorithms were shown to be reducible by choosing the current best matrix multiplication algorithm (i.e. Chandra's parallel version of Strassen's algorithm). By using Chandra's algorithm to compute the transitive closure of an n-node directed graph, efficient parallel algorithms were formulated for detecting the existence of negative cycles (in Section 2.2.2), finding the strongly connected components, verifying unilateral connectivity and acyclicness (in Section 3.2), each using $O(\log n(n^{2.81})/P)$ time units with $P \leq n^{2.81}/\log n$ processors, $S(P) = O(P/\log n)$ and $E(P) = O(1/\log n)$.

The construction of an $O(\log n)$ parallel algorithm for matrix multiplication using less than $O(n^{2.81}/\log n)$ processors remains an open problem whose solution would imply improvements on the corresponding sequential algorithm and many other matrix oriented problems.

The results in this thesis should provide a better understanding of the relationship of programs to machine organization offering new insights into the design of practical parallel computers.

REFERENCES

[1] Agerwala T. and B. Lint, "Communication in Parallel Algorithms for Boolean Matrix Multiplication," Proc. of the 1978 International Conf. on Parallel Processing, 1978, pp. 146-153.

[2] Arjomandi E., "A Study of Parallelism in Graph Theory," Ph.D. Dissertation TR 86, Dept. of Comp. Sci., U. of Toronto, 1975.

[3] Arjomandi E., "On Finding All Unilaterally Connected Components of a Digraph," Information Processing Letters, Vol. 5, May 1976, pp. 8-10.

[4] Backhous R.C. and B.A. Carre, "Regular Algebra Applied to Path-Finding Problems," J. Inst. Math. Appl., Vol. 15, pp. 161-186.

[5] Baer J.L., "A Survey of Some Theoretical Aspects of Multiprocessing," ACM Computing Surveys, Vol. 5, March 1973, pp. 31-80.

[6] Baudet G. and D. Stevenson, "Optimal Sorting Algorithms for Parallel Computers," IEEE Trans. on Computers, Vol. C-27, Jan. 1978, pp. 84-87.

[7] Berge C. and A. Chouila-Houri, Programming, Games, and Transportation Networks, John Wiley, New York, 1965, pp. 179.

[8] Best M.R., P. van Emde Boas, and H.W. Lenstra Jr., "A Sharpened Version of the Aanderaa-Rosebery Conjecture," IR ZW 30-74, Mathematisch Centrum, Amsterdam, 1974.

[9] Brent R., "The Paralell Evaluation of General Arithmetic Expressions," JACM Vol. 21, April 1974, pp. 201-206.

[10] Chandra A.K., "Maximal Parallelism in Matrix Multiplication," IBM Research Report RC 6193, 1976.

[11] Chen S.C., "Speedups of Iterative Programs in Multiprocessing Systems," Ph.D. Dissertation, Dept. of Comp. Sci., U. Of Illinois, Urbana, 1975.

[12] Crane B.A., "Path Finding with Associative Memory," IEEE Trans. on Computers, Vol. C-17, July 1968, pp. 691-693.

[13] Csanky L., "On the Parallel Complexity of Some Computational Problems," Ph.D. Dissertation, Comp. Sci. Divison, U. of California, Berkeley, 1974.

[14] Eckstein D.M. and D.A. Alton, "Parallel Graph Processing Using Depth-First Search," A Conf. on Theoretical Comp. Sci., U. of Waterloo, 1977, pp.21-29.

[15] Feng T.Y., (Editor) ACM Computing Surveys, Vol. 9, No. 1, March 1977, pp.3-129.

[16] Flynn M.J., "Some Computer Organizations and Their Effectiveness," IEEE Trans. on Computers, Vol. C-21, Sept. 1972, pp.948-960.

[17] Furman M.E., "Application of a Method of Fast Multiplication of Matrices in the Problem of Finding the Transitive Closure of a Graph," Soviet Math. Dokl., Vol. 11, No. 5 , 1970, pp.1252.

[18] Gentleman W.M., "Some Complexity Results for Matrix Computation on Parallel Processors," JACM, Vol. 25, Jan. 1978, pp.112-115.

[19] Goldschlager L.M., "Synchronous Parallel Computation," Ph.D. Dissertation TR 114, Dept. of Comp. Sci., U. of Toronto, 1977.

[20] Heller D., "A Survey of Parallel Algorithms in Numerical Linear Algebra," SIAM Review, Vol. 20, Oct. 1978, pp.740-777.

[21] Hirschberg D.S., "Parallel Algorithms for the Transitive Closure and the Connected Component Problems," Proc. of 8th Annual ACM Symposium on Theory of Computing, 1976, pp.55-57.

[22] Holt R.C. and E.M. Reingold, "On the Time Required to Detect Cycles and Connectivity in Graphs," Mathematical Systems Theory, Vol. 6, 1972, pp.103-106.

[23] Hyafil L. and H.T. Kung, "Parallel Algorithms for Solving Triangular Linear Systems with Small Parallelism," Dept. of Comp. Sci., Carnegie-Mellon U., Pittsburgh, Pa., 1974.

[24] Kirkpatrick D.G., "Determining Graph Properties from Matrix Representations," Proc. of 6th Annual ACM Symposium on Theory of Computing, 1974, pp.84-90.

[25] Kung H.T., and D. Stevenson, "A Software Technique for Reducing the Routing Time on a Parallel Computer with a Fixed Interconnecticn Network," in High Speed Computer and Algorithm Organization, Academic Press, 1977, pp. 423-433.

[26] Lawler E.L., Combinatorial Optimization: Networks and Matroids, Holt, Rinehart and Winston, 1976.

[27] Lee R., "Optimal Parallel Computations for SIMD Computer," Ph.D. Dissertation, Dept. of Comp. Sci., U. of Alberta, 1976.

[28] Levitt K.N. and W.H. Kautz, "Cellular Arrays for the Solution of Graph Problems," CACM, Vol. 15, Sept. 1972, pp. 789-801.

[29] Miranker W., "A Survey of Parallelism in Numerical Analysis," SIAM Review, Vol. 13, Oct. 1971, pp. 524-547.

[30] Munro I. and M. Paterson, "Optimal Algorithms for Parallel Polynomial Evaluation," JCSS, Vol. 7, 1973, pp. 189-198.

[31] Munro I., "Efficient Determination of the Transitive Closure of a Directed Graph," Information Processing Letters, Vol. 1, 1971, pp. 56-58.

[32] Ortega J. and R. Voigt, "Solution of Partial Differential Equations on Vector Computers," Proc. of the 1977 Army Numerical Analysis and Comp. Conf.

[33] Pease M.C., "The Indirect Binary n-Cube Microprocessor Array," IEEE Trans. on Computers, Vol. C-26, pp. 458-473, May 1977.

[34] Preparata F.P., "New Parallel-Sorting Schemes," IEEE Trans. on Computers, Vol. C-27, July 1978, pp. 669-673.

[35] Rivest R.L. and J. Vuillemin, "On Recognizing Graph Properties from Adjacency Matrices," Theor. Comp. Sci., Vol. 3, Dec. 1976, pp. 371-384.

[36] Sameh A.H. and D.J. Kuck, "Parallel Direct Linear System Solvers - A Survey," Mathematics and Computers in Simulation, XIX, 1977, pp. 272-277.

[37] Savage C.D., "Parallel Algorithms for Graph Theoretical Problems," Ph.D. Dissertation R-784, Dept. of Math., U. of Illinois, Urbana, 1977.

[38] Siegel H.J., "Interconnection Networks for SIMD Machines," Computer, Vol. 12, June 1979, pp.57-65.

[39] Strassen V., "Gaussian Elimination is Not Optimal," Numerische Mathematik, Vol. 13, 1969, pp.354-356.

[40] Sullivan H., T.R. Bashkow, and K. Klappholz, "A Large Scale Homogeneous Fully Distributed Parallel Machine," 4th Annual Symposium on Comp. Architecture, March 1977, pp.105-124.

[41] Thompson C. and H.T. Kung, "Sorting on a Mesh-Connected Parallel Computer," CACM, Vol. 20, April 1977, pp.263-270.

[42] Thurber K.J. and L.D. Wald, "Associative and Parallel Processors," ACM Computing Surveys, Vol. 17, Dec. 1975, pp.215-255.

[43] Trajan R., "Depth-First Search and Linear Graph Algorithms," SIAM J. Comput., Vol. 1, June 1972, pp.146-160.